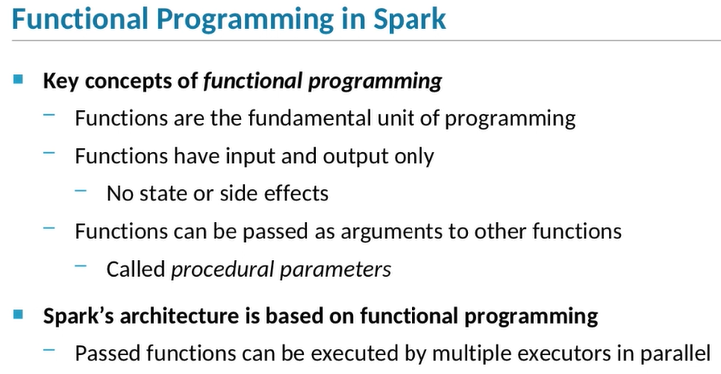
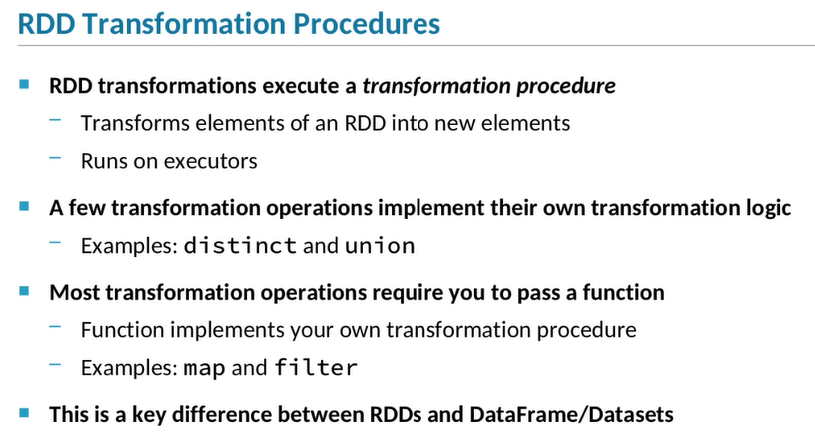
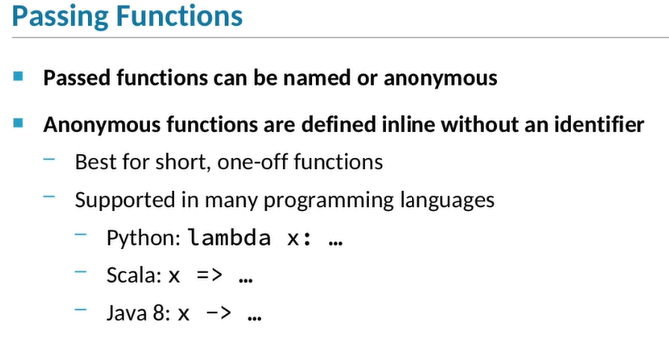
Writing and Passing Transformation Functions

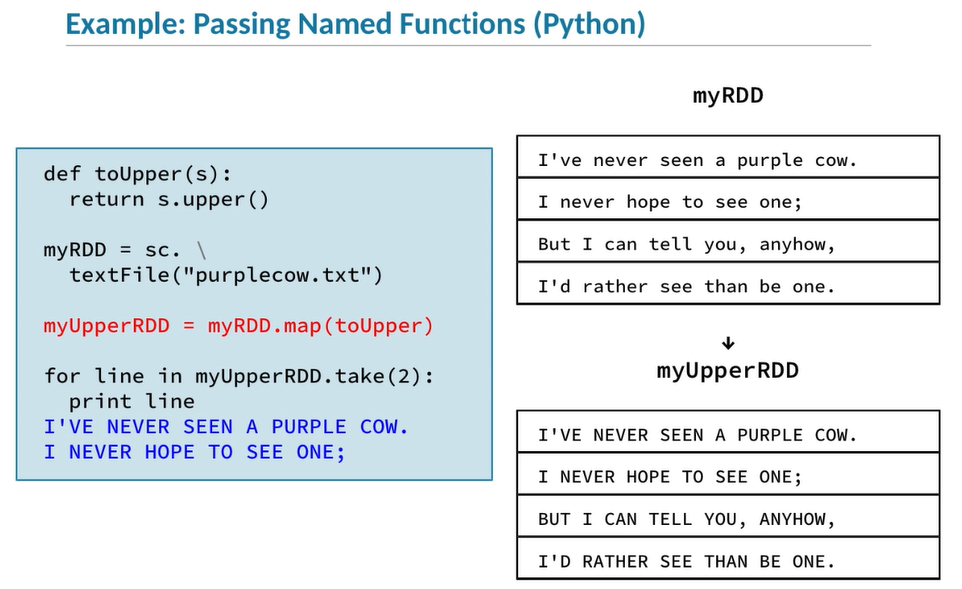


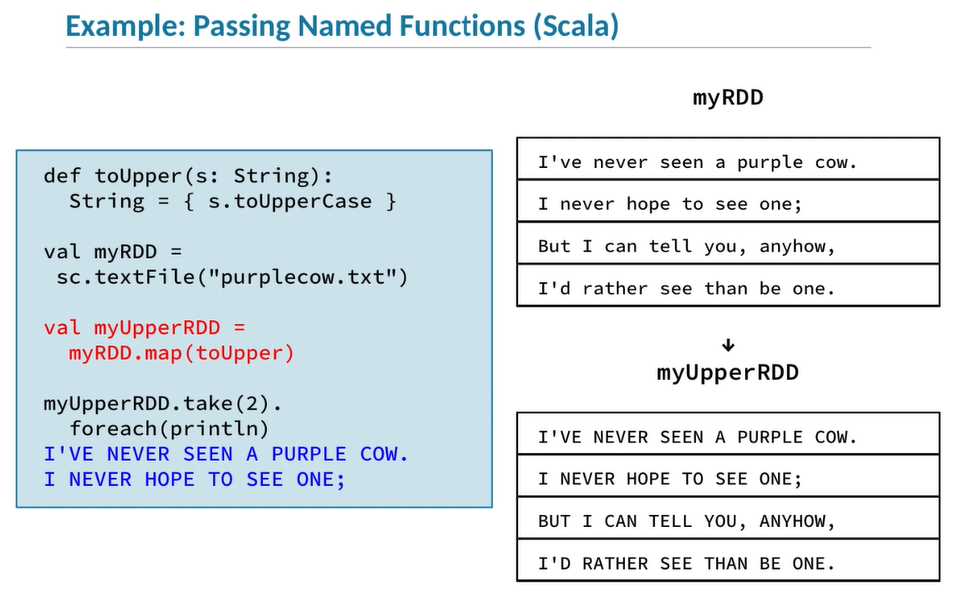


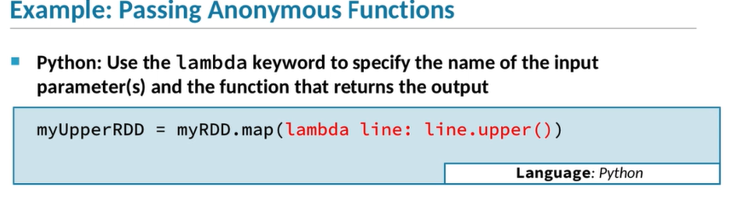


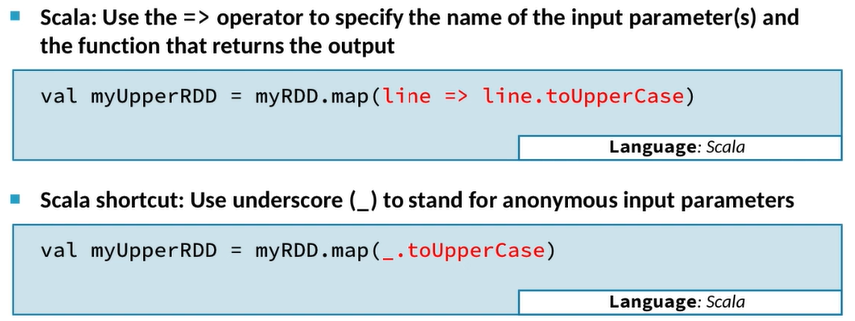
1. **Before you can start making full use of RDD transformations, you need to understand a bit about *functional programming*.**
2. Functional programming is a programming paradigm in which the fundamental unit is a function.
3. By definition, functions in functional programming take input values and return output values, but they don’t do anything else;
4. that is, they don’t affect state or have side effects.
5. Functional programming allows you to pass one function—or *procedure*—as a parameter to another.
6. The ability to pass a function is the key functional programming concept you need to use when transforming data in RDDs.
7. Functional programming is especially suited to distributed processing, because the passed function can be executed in parallel across multiple cluster nodes.
8. RDD transformations work by executing some logic—that is, some procedure or function—on the elements in an RDD to create new elements in a new RDD.
9. The transformation procedure is executed by your application's executors.
10. That's how Spark distributes RDD processing on a cluster.
11. In some cases, the transformation logic is implemented by the transformation operation itself.
12. For example, the distinct transformation procedure evaluates every element in an RDD to determine if any of the elements are duplicates,
13. and creates a new RDD with duplicate elements omitted.
14. You don't have to provide any additional instructions for the transformation to create the new RDD.
15. Similarly, the union operation also implements the transformation logic internally
16. in order to create a new RDD by appending the elements of an RDD to the elements of another.
17. However, most transformations don't work this way.
18. Most transformations require a procedural parameter: a transformation function.
19. To use these sorts of transformations, you write a function that defines the logic that transforms elements of one RDD into new elements.
20. When you call the transformation operation, you pass your function.
21. When the query is executed, the function you passed is applied to the elements in the RDD to transform them into elements in a new RDD.
22. For example, when you call the map operation, you pass a function that takes one RDD element as input and outputs a new element.
23. For the filter operation, you provide a function that takes an RDD element and returns a boolean value, indicating whether that element should be included in the new RDD.
24. The ability for the programmer to define the transformation logic is a key difference between RDD transformations and DataFrame and Dataset transformations.
25. With most DataFrame and Dataset transformations, the transformation logic is part of the transformation operation itself—
26. select returns the selected columns, where filters using the specified column expression, and so on.
27. This is one of the ways in which working directly with RDDs can provide you with more control than working with DataFrames and Datasets.
28. When you pass a function to a transformation, you can use either defined, named functions or anonymous functions, also known as lambdafunctions.
29. To use an anonymous function, you define the function inline, right in the transformation call.
30. You don’t need to assign a name to the function.
31. In Python, you use the lambda keyword to define anonymous functions.
32. In Scala, use the => operator, which some people refer to as a “hash rocket.”
33. Anonymous functions are used frequently in Spark.
34. When you define a one-off function—that is, when you are only going to use the function for one transformation—
35. it makes sense to keep your code succinct by defining the function inline.

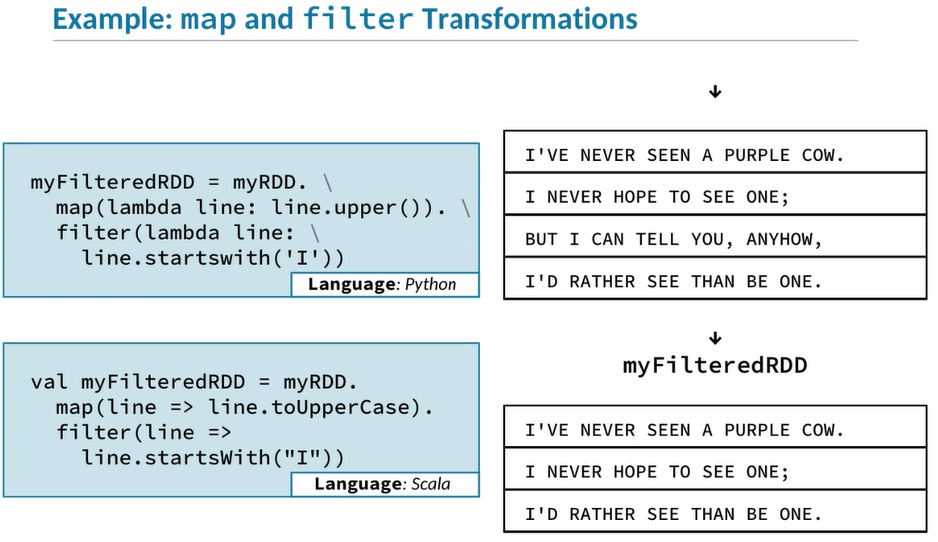
Writing and Passing Transformation Functions





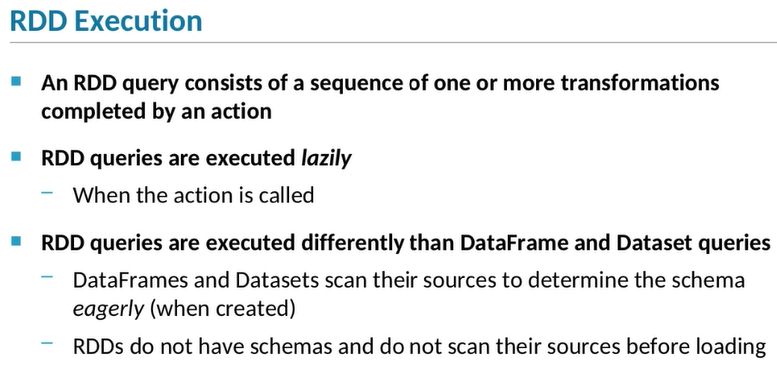


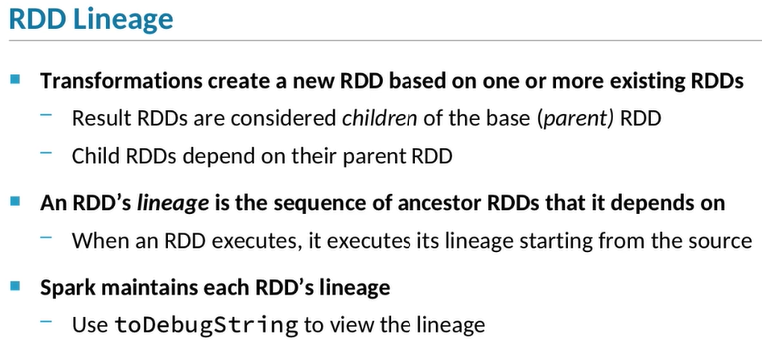


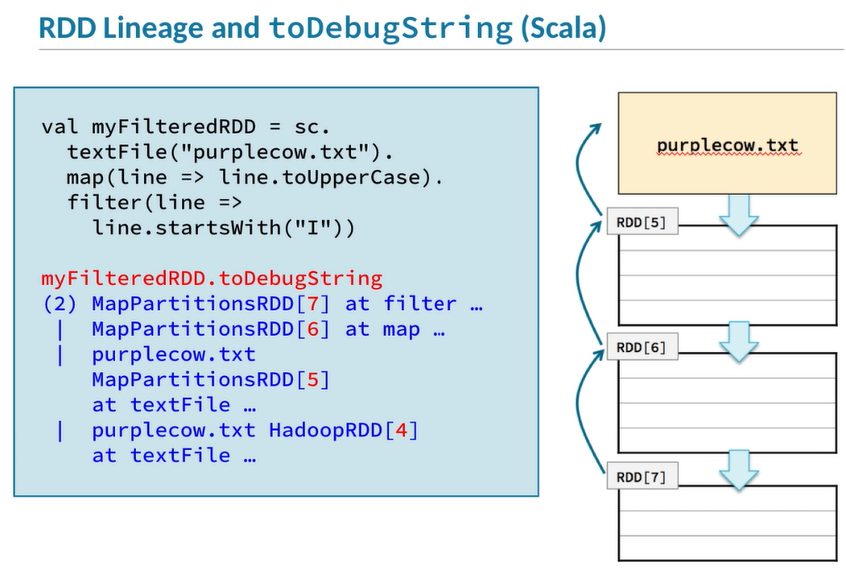


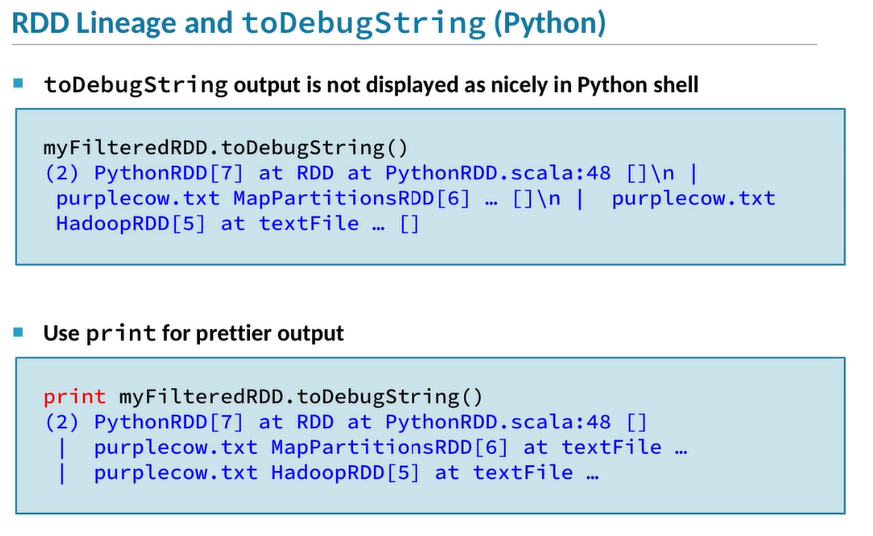
1. Let’s start with a Python example that passes a named function to a transformation.
2. We start by defining a simple function called toUpper, which takes a string and returns the same string, with all letters transformed to uppercase.
3. Next we create an RDD based on a small text file containing the "Purple Cow" poem.
4. Then we call the map operation and pass it the toUpper function we just defined.
5. The map operation is one of the most basic transformations.
6. It creates a new RDD by going through each element of the base RDD and passing that element to the function you specify.
7. In this example, we have an RDD of strings;
8. the map operation will call toUpper, passing each string in the base RDD in turn to the toUpper function.
9. In the final line of the example, we call the takeaction to return a list of the first two elements in the RDD.
10. We loop through the list, printing out the string elements, and as we can see, those strings have been correctly converted to uppercase.
11. This Scala example does exactly the same thing as the Python code we just looked at.
12. **It defines a toUpper function and uses it to convert all the lines from a text file to uppercase.**
13. The only difference between these examples is the Scala syntax for defining a function and for looping through the array returned by take.
14. Note that in Scala, the toUpper function requires a string as input parameter and returns a string output value.
15. Because Scala is strongly-typed, this means that you can only use this function when you are working with an RDD of strings.
16. If you use an RDD containing any other type of element, Spark will give a type mismatch error and will not attempt to execute the function.
17. Here are code snippets to demonstrate how to pass anonymous functions in Scala and Python.
18. These examples all do the same thing the last one did—they take a string element of an RDD and return the same string converted to uppercase—
19. but instead of defining a toUpper function, they use an anonymous function defined in-line.
20. The first snippet shows how Python uses the lambda syntax to define the function and pass it to the map operation.
21. Start with lambda, then specify the parameter that will be passed to the function—which in our example is a string we’re calling line—
22. then colon—and then what the function should return.
23. In this case, the function returns the uppercase version of the line parameter by calling the string’s upper function.
24. The second and third snippets are both Scala examples, showing two different syntaxes that can pass anonymous functions.
25. The first Scala snippet demonstrates the canonical Scala syntax.
26. We use the => operator—sometimes called *a hash-rocket*—to specify the parameter name—line—followed by the value the function returns—line.toUpperCase.
27. As you can see, this approach is very similar to using lambda in Python.
28. The third example shows a convenient syntactical shortcut.
29. Scala supports not just anonymous functions, but anonymous parameters too.
30. So here, instead of giving the string parameter a name like line, we can just use an underscore.
31. The first underscore refers to the first parameter that was passed to this anonymous function,
32. which in this case is the RDD element we want to transform to uppercase.
33. If our function took two parameters, we could use a second underscore to refer to the second parameter.
34. Here are Python and Scala code snippets demonstrating the map and filtertransformations, which use lambda functions, in sequence.
35. We start with the first RDD, which is based on purplecow.txt.
36. Then we call the map operation with a function to convert each element to uppercase, as we just saw a moment ago.
37. The next transformation in the chain is filter.
38. The filter transformation requires a function that takes as input an element of the RDD, and for each input element, returns a boolean value.
39. The operation will apply the function to each element; if the return value is true, it will include that element in the result RDD.
40. If it’s false, that element will be excluded.
41. In this example, our function tests each input string to see if it starts with the letter I.
42. All the strings in the RDD start with capital Iexcept the third, so that one is left out of the new, final RDD.

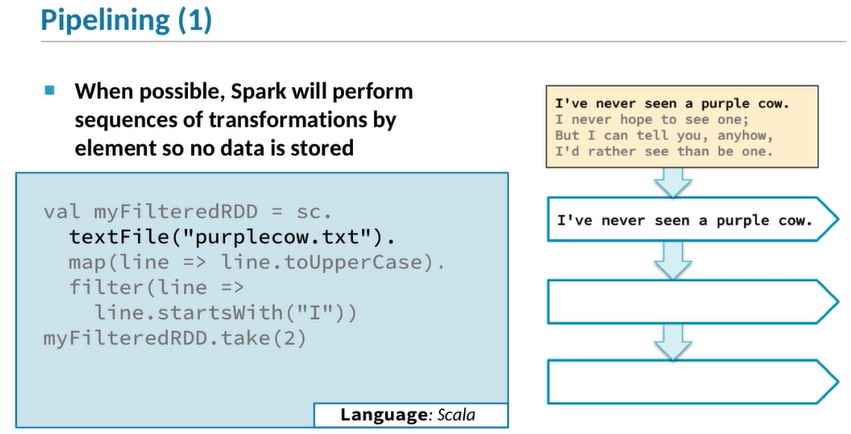
Transformation Execution





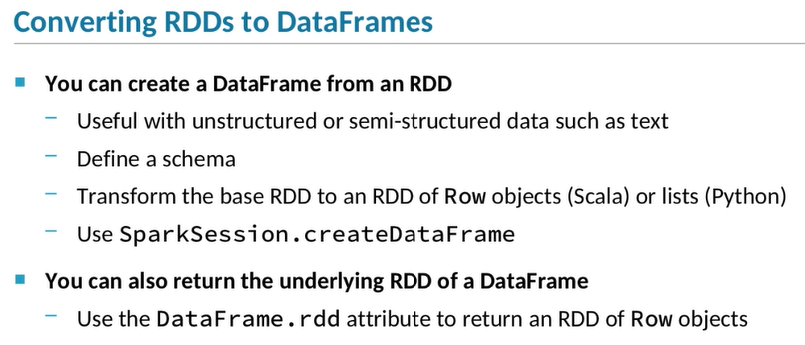


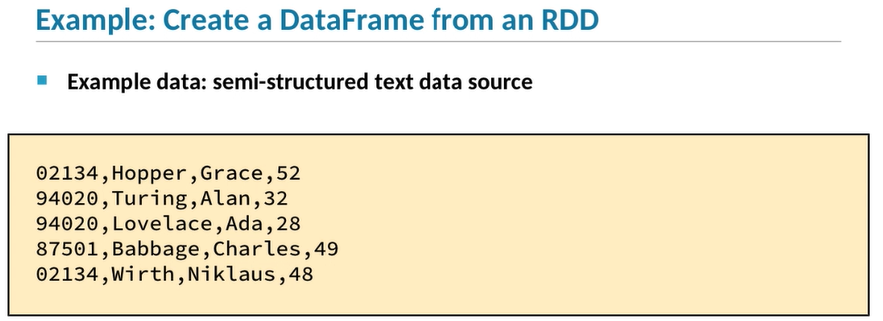


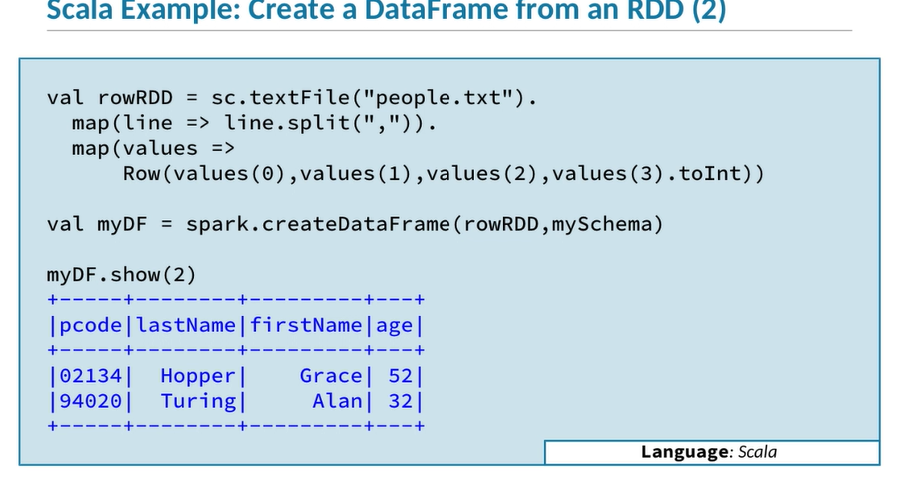


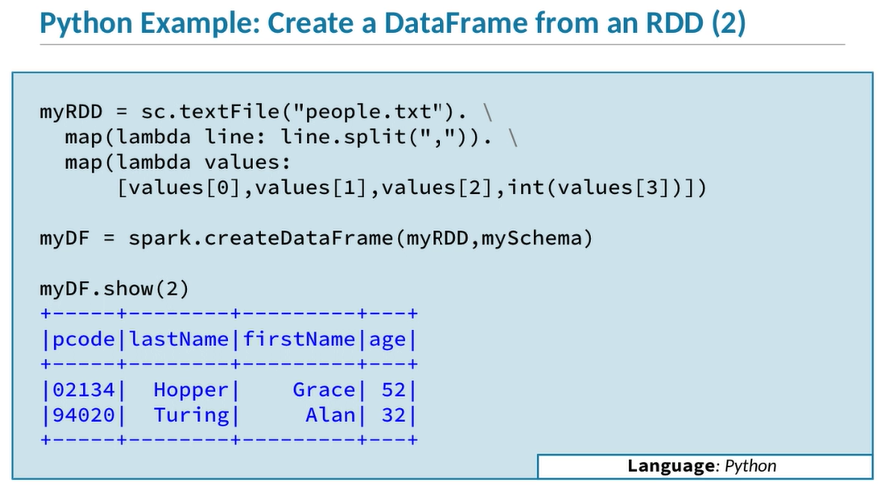
1. Now let's talk about how RDD transformations and actions are executed.
2. As you know, RDD transformations are often performed in series,
3. followed by an action that saves the data to files or returns a value to the driver.
4. This sequence of transformations and an action is sometimes called a *query*,
5. and executes much like DataFrame and Dataset queries do.
6. One difference, though, is that RDDs are always executed lazily
7. —that is, they don't load data from a data source and transform it
8. until an action is invoked.
9. DataFrames use both lazy and eager execution
10. —they scan their data source to define a schema eagerly when they are first created,
11. **but they don't actually execute until the query is triggered by an action.**
12. Because RDDs don't have schemas,
13. they don't need to do that initial scan when they are first created.
14. When a transformation creates a new RDD from an existing one,
15. the base RDD is referred to as a *parent*,
16. and the new one is called a *child*.
17. The relationship of each RDD to its parent,
18. going back to the base data source, is called the RDD’s *lineage*.
19. Spark keeps track of the lineage of each RDD.
20. The sequence of operations that are executed when a query is triggered
21. is determined by the lineage of the RDDs that make up the query.
22. The transformation of each RDD in the lineage
23. triggers the transformation of its parent,
24. which in turn transforms that RDD's parent,
25. continuing to the base data source.
26. To see the lineage for an RDD,
27. call the RDD’s toDebugString function.
28. This is an example of RDD.toDebugString output in Scala.
29. The first RDD in the displayed list is the final RDD in the lineage;
30. in this example, that’s RDD[7].
31. The next is the first one’s parent
32. and so on down to the first RDD, based on the original data source,
33. displayed last.
34. The Python output is exactly the same as with Scala,
35. except that in Python the RDD.toDebugStringoutput isn’t particularly readable.
36. If you use print RDD.toDebugString,
37. you get a much nicer representation of it,
38. as shown in the second code snippet.
39. Pipelining is another important Spark concept.
40. It’s easy to assume that when you execute an operation like myRDD = sc.textFile,
41. in reads the entire file into memory
42. —that is, that the data is actually stored within the RDD.
43. If that file was very large, though, that would be a problem,
44. because your application would probably run out of memory.
45. But that’s not what actually happens.
46. Whenever possible, Spark will pipeline sequences of transformations
47. so that it can operate on one element at a time.
48. Let’s look again at our map and filter example.
49. If you look at the last line you’ll see that it calls an action
50. —in this case, take(2).
51. You’ve already learned that an action on an RDD triggers the execution of all the transformations in the lineage
52. of that RDD back to the base.
53. So what actually happens when this sequence executes?
54. The first step is textFile(purplecow.txt);
55. at this point, Spark will read just the first line into myRDD.
56. Then it will execute the next transformation,
57. which is the map that calls line.toUpperCase.
58. It will perform that operation on the first line in the file.
59. Then it will execute the next transformation,
60. which in this case is a filter that checks if the line starts with uppercase letter I.
61. That returns true for the first element,
62. so that element is included in the return value Spark passes back to the take action.
63. The last operation is an action,
64. myFilteredRDD.take(2),
65. which returns the first two elements.
66. At this point, Spark returns the first element,
67. completing the lineage transformations on that element.
68. Then Spark will read the next line from purplecow.txt…
69. It will map that element to uppercase…
70. Then call the filter operation.
71. The element is included in the child RDD
72. because the filter procedure returns true for this element.
73. Now Spark returns the transformed second element to the driver.
74. In this example, we triggered execution of this query by calling take(2),
75. which means that Spark should only return two elements.
76. So at this point in the process, all the requested data has been returned to the driver,
77. and the query execution is now complete.
78. Because all the transformations in this query were pipelined,
79. Spark never loads and transforms the rest of the lines in the file.
80. Spark's pipelining feature lets you work with very large files
81. without having to worry about running out of memory.
82. It also improves performance because it only processes the data required by the query.
83. **Although DataFrames and RDDs represent data in different ways, you can use them together in the same application.**
84. You can create DataFrames from RDDs and vice versa.
85. For example, you might want to convert an RDD to a DataFrame if your RDD data source contains unstructured or semi-structured data, such as text.
86. To create a DataFrame from an RDD, you start by defining a schema for the data.
87. This is necessary because RDDs are inherently unstructured and DataFrames are inherently structured.
88. In order for the DataFrame to apply a schema to the RDD's data, the data needs to be in structured form.
89. So, your next step is to structure your data by transforming whatever type was created from reading the base data source into Row objects.
90. Python offers a shortcut;
91. if your RDD contains lists, Spark will treat them as Row objects.
92. The final step to convert your RDD of lists or rows to a DataFrame is to pass the RDD to the SparkSession createDataFrame function.
93. DataFrames are actually implemented using RDDs, so every DataFrame has an underlying RDD of Row objects.
94. You can work directly with that RDD using the DataFrame.rdd attribute.
95. Sometimes this makes sense if your use case requires a complex transformation that can’t be easily expressed with a DataFrame query.
96. In some ways, RDDs are more flexible than DataFrames because you can write your own arbitrarily complex transformation functions.
97. This comes at a cost, however;
98. whichever part of your processing uses passed functions can’t be optimized by the Catalyst optimizer.
99. Let’s look at an example showing how to create a DataFrame from an RDD.
100. We’ll start with a data file called people.txt, containing text where each line includes a postal code, last name, first name, and age, separated by commas.
101. This format, of course, is a CSV file, which could be easily read into a DataFrame.
102. We’re using this format for demo purposes because it is easy to work with.
103. But the format could be much more complicated than this, and the principles we will demonstrate here are the same, regardless.
104. Let’s start with a Scala example.
105. As we said, the first step is to define a schema describing the structure of the new DataFrame.
106. Here we define a schema—that is, a StructTypeobject—that contains three string columns called pcode, lastName, and firstName, and an integer age column.
107. This slide continues with the Scala example.
108. Here we create an RDD based on the people.txtdata file.
109. The base RDD contains strings, where each string corresponds to a line in the file.
110. We can’t just use the strings as is, though.
111. We need to structure the data into rows.
112. So, the next line splits each string by comma, resulting in an array containing each field value as a string.
113. The final step in transforming the RDD is to create Row objects using the values in the split array.
114. For each array element, we create a Row object, passing the array values.
115. The pcode, firstName, and lastName fields are passed along as strings,
116. but the schema for the fourth column, age, requires that the corresponding Row value be an integer.
117. So on the fourth element, we parse the string to an integer and pass the result.
118. Now that we have a Row RDD and a schema, we can use those to create a new DataFrame.
119. When we show the data in the DataFrame, it is correctly structured according to the schema.
120. The Python version of this example is very similar to the Scala version.
121. Again, we start by creating a schema describing the structure of the new DataFrame we plan to create.
122. Then we read the people.txt file and map it to a structured form.
123. Note one small difference from the Scala version:
124. instead of creating an RDD of Row objects, we use a list of values.
125. You could use Row objects as well, but because Python is loosely-typed, we can use a list instead as a shortcut.
126. Our next example extends the previous one to show the reverse process, getting an RDD from a DataFrame.
127. First, we retrieve the Row RDD that underlies the DataFrame.
128. Then we loop through the first two elements of that RDD, displaying each.
129. Note that the structure of the data in the rows is preserved, even though the RDD itself isn’t structured.

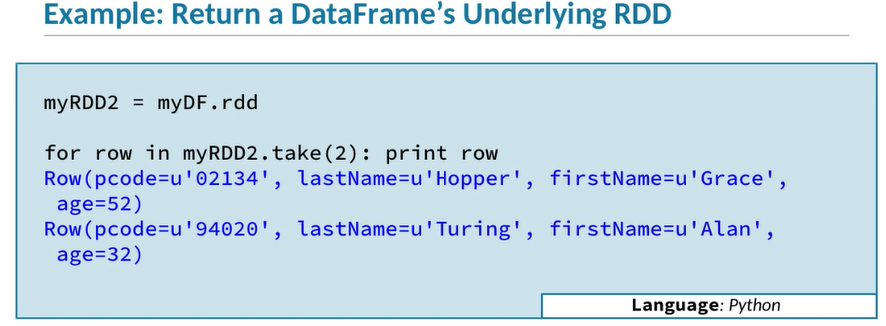
Converting Between RDDs and DataFrames

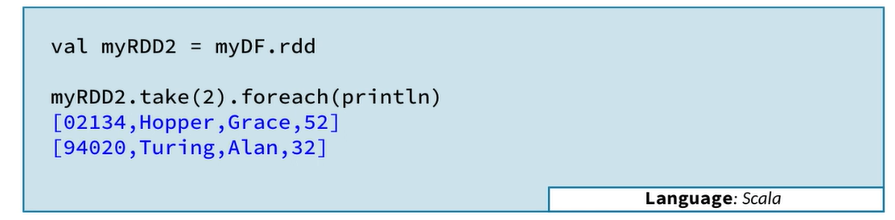












1. **Although DataFrames and RDDs represent data in different ways, you can use them together in the same application.**
2. You can create DataFrames from RDDs and vice versa.
3. For example, you might want to convert an RDD to a DataFrame if your RDD data source contains unstructured or semi-structured data, such as text.
4. To create a DataFrame from an RDD, you start by defining a schema for the data.
5. This is necessary because RDDs are inherently unstructured and DataFrames are inherently structured.
6. In order for the DataFrame to apply a schema to the RDD's data, the data needs to be in structured form.
7. So, your next step is to structure your data by transforming whatever type was created from reading the base data source into Row objects.
8. Python offers a shortcut;
9. if your RDD contains lists, Spark will treat them as Row objects.
10. The final step to convert your RDD of lists or rows to a DataFrame is to pass the RDD to the SparkSession createDataFrame function.
11. DataFrames are actually implemented using RDDs, so every DataFrame has an underlying RDD of Row objects.
12. You can work directly with that RDD using the DataFrame.rdd attribute.
13. Sometimes this makes sense if your use case requires a complex transformation that can’t be easily expressed with a DataFrame query.
14. In some ways, RDDs are more flexible than DataFrames because you can write your own arbitrarily complex transformation functions.
15. This comes at a cost, however;
16. whichever part of your processing uses passed functions can’t be optimized by the Catalyst optimizer.
17. Let’s look at an example showing how to create a DataFrame from an RDD.
18. We’ll start with a data file called people.txt, containing text where each line includes a postal code, last name, first name, and age, separated by commas.
19. This format, of course, is a CSV file, which could be easily read into a DataFrame.
20. We’re using this format for demo purposes because it is easy to work with.
21. But the format could be much more complicated than this, and the principles we will demonstrate here are the same, regardless.
22. Let’s start with a Scala example.
23. As we said, the first step is to define a schema describing the structure of the new DataFrame.
24. Here we define a schema—that is, a StructTypeobject—that contains three string columns called pcode, lastName, and firstName, and an integer age column.
25. This slide continues with the Scala example.
26. Here we create an RDD based on the people.txtdata file.
27. The base RDD contains strings, where each string corresponds to a line in the file.
28. We can’t just use the strings as is, though.
29. We need to structure the data into rows.
30. So, the next line splits each string by comma, resulting in an array containing each field value as a string.
31. The final step in transforming the RDD is to create Row objects using the values in the split array.
32. For each array element, we create a Row object, passing the array values.
33. The pcode, firstName, and lastName fields are passed along as strings,
34. but the schema for the fourth column, age, requires that the corresponding Row value be an integer.
35. So on the fourth element, we parse the string to an integer and pass the result.
36. Now that we have a Row RDD and a schema, we can use those to create a new DataFrame.
37. When we show the data in the DataFrame, it is correctly structured according to the schema.
38. The Python version of this example is very similar to the Scala version.
39. Again, we start by creating a schema describing the structure of the new DataFrame we plan to create.
40. Then we read the people.txt file and map it to a structured form.
41. Note one small difference from the Scala version:
42. instead of creating an RDD of Row objects, we use a list of values.
43. You could use Row objects as well, but because Python is loosely-typed, we can use a list instead as a shortcut.
44. Our next example extends the previous one to show the reverse process, getting an RDD from a DataFrame.
45. First, we retrieve the Row RDD that underlies the DataFrame.
46. Then we loop through the first two elements of that RDD, displaying each.
47. Note that the structure of the data in the rows is preserved, even though the RDD itself isn’t structured.